

# 第五章 类:初始化和清除

面向对象程序设计(C++)



## 5.3 初始化和清除

---

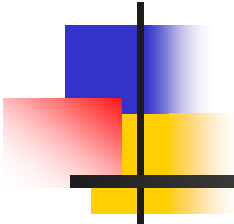
- ➡ 构造函数
- ➡ 析构函数
- ➡ 何时被调用
- ➡ 默认构造函数与重载构造函数



## 5.3.1 安全性要求

- 正确地初始化和清除对象是保证程序安全性的关键问题!

```
int main() {  
  
    Stash intStash;  
    intStash.initialize(sizeof(int)); //初始化  
  
    for(int i = 0; i < 100; i++){  
        intStash.add(&i);  
    }  
    .....;  
    intStash.cleanup();           // 清除  
}
```



## 5.3.2 构造函数

---



## 5.3.2.1 构造函数：确保初始化

---

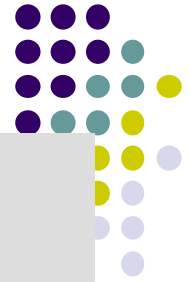
- 类的特殊成员函数，编译器在创建对象时**自动**调用该函数。通常做一些初始化动作。以保证同一个类的对象具有一致性。
- 构造函数跟类同名，可以带参数，没有返回值。
  - 跟别的成员函数没有名字冲突；
  - 编译器总能知道调用哪一个函数；

## 例：带构造函数的Stash

```
class Stash {
    int size;    // Size of each space
    ...;
public:
    Stash(int size); // 构造函数，跟类同名,无返回值
    int add(void* element);
    ...;
    void cleanup();
};

Stash::Stash(int sz) {
    size = sz;
    quantity = 0;
    storage = 0;
    next = 0;
}
```

## 自动初始化Stash对象



```
int main() {  
  
    //创建对象时自动调用构造函数  
    Stash intStash1(2), intStash2(2);  
  
    for(int i = 0; i < 100; i++){  
        intStash1.add(&i);  
    }  
    .....;  
    intStash1.cleanup();           // 清除  
}
```

下面代码输出?

```
count=0
```

```
class C{  
    public:  
        C() {cout<<++count<<endl;}  
};
```

```
C obj[100]
```





## 说明

- 构造函数的名字必须与类的名字相同;
- 构造函数不允许指明返回类型, 也不允许返回一个值;
- 构造函数应声明为public (但不是必须), 否则无法创建对象。

```
class Stash{  
    private:  
        Stash(int sz){...};  
    ...  
};  
  
void main(){  
    Stash  s(10); // Error! can't access private member  
}
```



```
class Student{  
    private:  
        unsigned id;  
        Student();  
    public:  
        Student(unsigned ID){id=ID;}  
}
```

//希望强制id

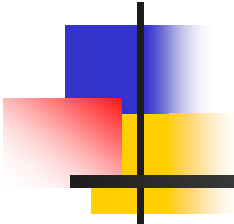


## 说明(续)

- 可以重载构造函数;

```
class Stash{
public:
    Stash() {size = 1; ...; }
    Stash(int sz){...}
    ...
};

void main(){
    Stash  charStash;
    Stash  intStash(2);
    Stash  intStash(10, 2); // error,参数不匹配!
}
```



## 5.3.3 析构函数

---



### 5.3.3.1 析构函数：确保清除

---

- **析构函数：类的特殊成员函数，在撤销对象时自动调用该函数。通常做一些撤销对象前的回收工作。**
- **析构函数不带参数，没有返回值；不能够重载。**
- **析构函数必须是public函数。**

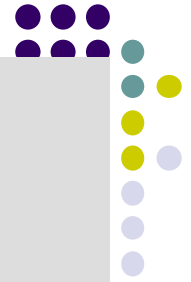
## 例: String类



```
Class String{
private:
    char * str;
public:
    String(char * s); // 构造函数，跟类同名
    String(unsigned int sz); // 重载构造函数
    ...;
    ~String(); // 析构函数，类名前面加上一个~
};

String::~~String(){
    delete str;
}

String::String(unsigned int sz){ // 不能有返回值;
```



```
if(sz > 0)  str= new char[sz];  
    else  str = new char [100];  
}
```

```
String::String(char * s){ // 不能有返回值;  
    int len;  
    len = strlen(s);  
    if (len<100) len =100;  
    str = new char[len];  
    strcpy(str, s);  
}
```

```
void main(){  
    String  Paper_Tiger("U.S.A.");  
} //退出主程序前撤销对象，调用析构函数
```



## 5.3.4 何时被执行？

---





## 5.3.4.1 何时被执行？

- 构造函数：当对象被创建时，调用构造函数；
- 析构函数：当对象被撤销时，调用析构函数。

// 此例说明何时调用构造函数和析构函数

// 类的定义

```
class Tree {  
    int height;  
public:
```

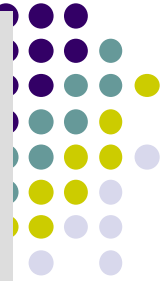
```
    Tree(int initialHeight); // 构造函数  
    ~Tree();                 // 析构函数
```

```
    void grow(int years);  
    void printsize();
```

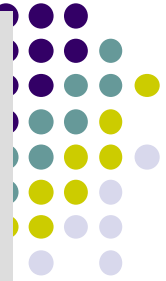
```
};
```

## //类的实现

```
Tree::Tree(int initialHeight) {  
    height = initialHeight; printsize();  
}  
Tree::~~Tree() {  
    cout << "inside Tree destructor" ;  
  
}  
  
void Tree::grow(int years) {  
    height += years;  
}  
  
void Tree::printsize() {  
    cout << "Tree height is " << height << endl;  
}
```



```
int main() {  
    cout << "before opening brace" << endl;  
    {  
        Tree t(12); //生成 t  
        cout << "after Tree creation" << endl;  
        t.grow(4);  
        cout << "before closing brace" << endl;  
    } // 此时撤销t;  
    cout << "after closing brace" << endl;  
} ///:~
```



```
int main() {  
    cout << "before opening brace" << endl;  
    {  
        Tree t(12); //生成 t  
        cout << "after Tree creation" << endl;  
        t.grow(4);  
        cout << "before closing brace" << endl;  
    } // 此时撤销t;  
    cout << "after closing brace" << endl;  
} ///:~
```

before opening brace

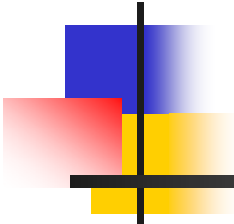
Tree height is 12

after Tree Creation

before closing brace

inside Tree destructor //执行析构函数

after closing brace



## 5.3.5 完整的类

---



## 5.3.5.1 完整的类

---

- 完整的类通常具有的特征：
  - ✓数据抽象：数据成员&成员函数
  - ✓实现隐藏；(访问控制)
  - ✓自动初始化和清除；(构造函数和析构函数)

## 例：类Stash

//: C06:Stash2.h

**// With constructors & destructors**

**#ifndef STASH2\_H**

**#define STASH2\_H**

**class Stash {**

**int size;        // Size of each space**

**int quantity; // Number of storage spaces**

**int next;       // Next empty space**

**// Dynamically allocated array of bytes:**

**unsigned char\* storage;**

**void inflate(int increase);**

**public:**





```
Stash(int size);  
~Stash();  
int add(void* element);  
void* fetch(int index);  
int count();  
};  
#endif // STASH2_H ///:~
```

### **//: C06:Stash2.cpp**

```
// Constructors & destructors  
#include "Stash2.h"  
#include "../require.h"  
#include <iostream>  
#include <cassert>  
using namespace std;  
const int increment = 100;
```





```
Stash::Stash(int sz) { //构造函数
```

```
    size = sz;
```

```
    quantity = 0;
```

```
    storage = 0;
```

```
    next = 0;
```

```
}
```

```
int Stash::add(void* element) {
```

```
    if(next >= quantity)  inflate(increment);
```

```
    // Copy element into storage,
```

```
    // starting at next empty space:
```

```
int startBytes = next * size;
```

```
unsigned char* e = (unsigned char*)element;
```

```
for(int i = 0; i < size; i++)
```

```
    storage[startBytes + i] = e[i];
```

```
next++;
```

```
return(next - 1); // Index number
```




```
}
```

```
void* Stash::fetch(int index) {  
    require(0 <= index, "Stash::fetch (-)index");  
    if(index >= next)  
        return 0; // To indicate the end  
  
    // Produce pointer to desired element:  
    return &(storage[index * size]);  
}
```

```
int Stash::count() {  
    return next; // Number of elements in CStash  
}
```

```
void Stash::inflate(int increase) {
```



```
    require(increase > 0, "Stash::inflate zero or
negative increase");
    int newQuantity = quantity + increase;
    int newBytes = newQuantity * size;
    int oldBytes = quantity * size;
    unsigned char* b = new unsigned char[newBytes];
    for(int i = 0; i < oldBytes; i++)
        b[i] = storage[i]; // Copy old to new
    delete [ ](storage);    // Old storage
    storage = b; // Point to new memory
    quantity = newQuantity;
}
```


```
Stash::~~Stash() {
    if(storage != 0) {
        cout << "freeing storage" << endl;
    }
}
```



```
        delete [ ]storage;  
    }  
} ///:~
```

### //: C06:Stash2Test.cpp

```
#include "Stash2.h"  
#include "../require.h"  
#include <fstream>  
#include <iostream>  
#include <string>  
using namespace std;  
  
int main() {  
    Stash intStash(sizeof(int));  
    for(int i = 0; i < 100; i++)  
        intStash.add(&i);  
}
```



```
for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = " << *(int*)
        intStash.fetch(j) << endl;
const int bufsize = 80;
Stash stringStash(sizeof(char) * bufsize);
ifstream in("Stash2Test.cpp");
assure(in, " Stash2Test.cpp");
string line;
while(getline(in, line))
    stringStash.add((char*)line.c_str());
int k = 0;
char* cp;
while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
        << cp << endl;
} ///:~
```



## 5.3.6 默认的构造函数

- 不带任何参数的构造函数。

```
class Y{  
    ... ;  
}  
Y objy; // OK, Why?
```

**如果没有提供任何构造函数，编译器会创建一个默认的构造函数，但这个函数所实现的功能很少是我们所期望的。因此，应明确定义默认构造函数。**



## 5.3.6 默认的构造函数(续)

- 如果定义了构造函数，但没有无参构造函数，则创建对象时，若不带参数，将会出错。

```
class Y{  
    Y(int sz){...}  
    ... ;  
}
```

```
Y objy; // error,参数类型不匹配
```



## 5.3.7 显式调用析构函数

- 如果只想执行析构函数中的执行的操作，而不释放对象的空间，则可以显式调用析构函数。

```
//执行系统的析构函数，不释放对象的空间。  
pb->~String();
```





## 5.3.8 拷贝构造函数

```
Student stu2(stu1);
```

- 不提供拷贝构造函数，则编译器自动生成

定义方法：

```
Student(Student&);  
Student(const Student&)
```

可以多参数，但第二个参数开始必须有默认值

重载赋值操作符=?



## 5.3.8 拷贝构造函数

---

- 浅拷贝指向相同空间

```
class X
{
private:
    int i;
    int *pi;
public:
    X(): pi(new int){ }
    X(const X& copy): i(copy.i), pi(copy.pi){ }
};
```



## 5.3.8 拷贝构造函数

---

- 深拷贝是被拷贝对象的克隆

```
class X
{
private:
    int i;
    int *pi;
public:
    X(): pi(new int){ }
    X(const X& copy): i(copy.i), pi(new int(*copy.pi)){ }
};
```



## 5.3.8 拷贝构造函数

- 可以使用delete指定不生成拷贝构造函数和赋值

```
class Student
{
public:

    Student(const Student& p) = delete;

    Student& operator=(const Student& p) = delete;

private:
    unsigned id;
    string name;
};
```



## 5.3.8 一个例子

```
class Student{
public:
    Student(){}
    Student(const Student& p)
    {
        cout << "Copy Constructor" << endl;
    }

    Student& operator=(const Student& p)
    {
        cout << "Assign" << endl;
        return *this;
    }

private:
    unsigned id;
    string name;
};
```



## 5.3.8 一个例子

```
void f(Student p){ return; }
```

```
Student f1(){  
    Student p;  
    return p;  
}
```

```
int main(){  
    Student p;  
    Student p1 = p;  
    Student p2;  
    p2 = p;  
    f(p2);  
    p2 = f1();  
    return 0;  
}
```



## 5.3.8 一个例子

```
void f(Student p){ return; }
```

```
Student f1(){  
    Student p;  
    return p;  
}
```

```
int main(){  
    Student p;  
    Student p1 = p;  
    Student p2;  
    p2 = p;  
    f(p2);  
    p2 = f1();  
    return 0;  
}
```

**Copy Constructor**  
**Assign**  
**Copy Constructor**  
**Assign**



## 禁止通过传值返回对象

```
class C{  
    public:  
        C();  
    private:  
        C(C&);  
};  
  
void f(C);    //error  
C g();        //error
```





## 5.3.9 转型构造函数

---

- 当构造函数只有一个参数时, 编译时有一个缺省操作: 将该构造函数对应数据类型的数据转换为该类对象

```
class Age
{
public:
    Age(int a){age = a;}
private:
    int age;
};
```



## 5.3.9 转型构造函数

- 隐式类型转换

```
class Student
{
public:
    Student(){}
    Student(char * n){name = n;}
private:
    char * name;
};

void fun(Student s); //函数声明

char * name = "Harry Potter";
fun(name);
```



## 5.3.9 转型构造函数

- 关闭隐式类型转换

```
class Student
{
public:
    Student(){}
    explicit Student(char * n){name = n;}
private:
    char * name;
};
```

```
void fun(Student s); //函数声明
```

```
char * name = "Harry Potter";
fun(name);
```



## 小结

---

- **安全性非常重要。**
- **构造函数和析构函数是确保安全性的两个重要机制。**